UvA Matching 2016:
Problem Solving with Prolog (BSc KI)

Ulle Endriss
Institute for Logic, Language and Computation
University of Amsterdam

[https://staff.science.uva.nl/u.endriss/teaching/uva-matching/]
Plan for this Module

• Big picture:
  – What is *Artificial Intelligence* (AI)?
  – What is *problem solving* and why is it so central to AI?
  – How can *logic programming* help?
  – How does all of this relate to the *BSc KI* at the UvA?

• And then for the actual topic of this lecture:
  – Strategies for solving *Towers of Hanoi*
  – Implementing those strategies in (simplified!) *Prolog*
Artificial Intelligence

What is AI? People do not agree on how to answer this.

My favourite answer:

*AI is whatever AI researchers do.*

This is true, but not helpful to you right now. So maybe this:

*AI is about getting machines to perform tasks that we tend to associate with “intelligence” when performed by humans.*

More specifically:

- no need for the machine to *be* intelligent (philosophy)
- no need to agree what “intelligence” *is* exactly (psychology)
- no need to focus *only* on things humans are good at
Problem Solving

AI is about getting machines to perform tasks . . .

At the core of such (possibly physical) tasks there often are rather abstract problems we need to solve. A solution often requires us to find a sequence of actions to achieve a given goal:

- finding a path from A to B on a street map
- computing a sequence of electronic signals for a robot to do X
- solving Rubik’s Cube
Logic Programming

*Prolog* is a programming language that is particularly useful for problem solving. It is based on an idea called *logic programming*. 
Prolog and the BSc KI

In the BSc KI course *Logisch Programmeren en Zoektechnieken* you will learn how to program in Prolog. Many links to other courses:

- Foundations: *Inleiding Logica* and *Computationele Logica*
- *Taaltheorie en Taalverwerking* (grammar of natural language)
- *Zoeken, Sturen en Bewegen* (chess-playing robots)
- *Kennissystemen* (large knowledge bases)
- Other programming courses (basic principles)
Towers of Hanoi

Move all disks from the leftmost to the rightmost peg, whilst:

- moving only one disk at a time, and
- never placing a disk on top of a smaller disk
Problem Representation

We don’t want to build a robot that can physically move the disks (not today, anyway). We just want to find a general method that tells us in which order to move the disks.

First, we need to find an appropriate level of representation . . .
Graphical Representation
Prolog-internal Representation

Game = \[[3, 4, 5, 6, 7, 8, 9], [9], [1, 2, 9]\]
Representation in Terms of Bits

01001010 11010010 11101010 0010111 10100101 00101010
11101010 00101100 00001000 1101010 11010101 11010101
00101010 01011101 11010101 0010101 10100110 10100100
00101011 10100111 01011111 1010010 10101110 10101110
10000010 10101000 00101010 0001101 11010010 00001011
00010100 10001001 11101000 0111101 01010110 11110010
00101110 10100010 11010100 1010100 01111011 11001000
11101011 10100010 00110101 1101000 11100000 11000011
01110100 01001100 10100010 1110010 00101010 11110101
00101011 11101010 00001100 0101010 10101011 11100101
01010001 01010100 11100001 0101011 11110101 11000010
11101000 11100010 00101110 1010100 11000000 10001011
Physical Representation on the Computer
Game Console

You cannot learn how to program in just one day. And anyway, we want to focus on *high-level ideas*, not on low-level nitty gritty.

All the low-level stuff is taken care of by the *ToH Game Console*, a Prolog program providing a few simple *predicates* ("commands") for you to manipulate the towers.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init(+N)</td>
<td>create a game with N disks (with N ∈ {1, 2, 3, 4, …})</td>
</tr>
<tr>
<td>move(+A,+B)</td>
<td>move the top disk on peg A to B (A, B ∈ {1, 2, 3})</td>
</tr>
<tr>
<td>top(+A,-D)</td>
<td>given peg A, return top disk D on A (if any)</td>
</tr>
<tr>
<td>empty(+A)</td>
<td>check whether peg A is empty</td>
</tr>
</tbody>
</table>

Here ‘+’ means that you *provide* a value and ‘−’ means that you *receive* a value (by using a variable). Example:

```
?- top(1,X).
X = 3.
```
First Solution

Focus on the case of 3 disks for now.

To solve the game, you have to
move the top disk on peg 1 to peg 3, and then
move the top disk on peg 1 to peg 2, and then
move the top disk on peg 3 to peg 2, and then...

In Prolog this is implemented as follows:

sillySolve :-
    move(1,3), move(1,2), move(3,2), move(1,3),
    move(2,1), move(2,3), move(1,3).

This will work. Type into Prolog (with the Game Console loaded):

?- init(3), sillySolve.

Good solution?
Second Solution

What are the possible moves available at a given time?

- move *disk 1* (the smallest disk) “to the right” [1R]
- move *disk 1* (the smallest disk) “to the left” [1L]
- make the only possible move *not involving disk 1* [N1]
  (note that N1 is not possible in the initial or final configuration)

Observations:

- never iterate 1R/1L (can achieve same effect with one move)
- never iterate N1 (amounts to undoing previous move)

Thus:

- should alternate 1R/1L with N1

But which one, 1R or 1L? Leap of faith: let’s just try with 1R.
The Iterative Solution

To summarise, our algorithm is this:

(1) Perform **1R** (move disk 1 to the right, from peg 1 to peg 2).

(2) **Repeat** until the final configuration is reached:

   (a) Perform **N1** (only possible move not involving disk 1).

   (b) Perform **1R** (move disk 1 one peg “to the right”).

*This works!* (at least when the number $n$ of disks is even)

If $n$ is odd, use **1L** instead of **1R**.

This is pretty cool. But hard to understand *why* it actually works.
The Iterative Solution in Prolog

This is not so easy to implement and we’ll brush over some details.

\[
\text{move1R :}
\begin{align*}
& \text{\% to perform 1R:} \\
& \text{\% find peg of disk 1} \\
& \text{\% move disk 1 one peg to the right} \\
\end{align*}
\]

\[
\text{moveN1 :}
\begin{align*}
& \text{\% to perform N1:} \\
& \text{\% find peg of disk 1} \\
& \text{\% find peg of next smallest top disk} \\
& \text{\% find name of third peg} \\
& \text{\% move second smallest disk to third peg} \\
\end{align*}
\]

\[
\text{itSolve :}
\begin{align*}
& \text{\% to solve the game:} \\
& \text{\% perfom 1R, then finish off} \\
\end{align*}
\]

\[
\text{finish :}
\begin{align*}
& \text{\% finished if: pegs 1 & 2 empty} \\
\end{align*}
\]

\[
\text{finish :}
\begin{align*}
& \text{\% to finish:} \\
& \text{\% perform N1, then 1R, then finish off} \\
\end{align*}
\]
Third Solution

Wanted: method to move a tower of $n$ disks from peg A to peg B.

Solution:

To move a tower of $n$ (with $n > 1$) disks from A to B, simply

1. move $n-1$ disks from A to C,
2. move the largest disk from A to B, and
3. move $n-1$ disks from C to B

For the case of a tower with just 1 disk, we know what to do.

This works. Magic?
**Two Auxiliary Predicates**

We will need two auxiliary predicates that take care of some very basic stuff for us that is related to the manipulation of numbers.

getPredecessor(+N,-P): Given a natural number N, return that number’s predecessor $P := N - 1$. Example:

?- getPredecessor(10,X).
   X = 9.

getThirdPeg(+A,+B,-C): Given the names of pegs A and B, return that of the third peg C (names are 1, 2, and 3). Example:

?- getThirdPeg(3,1,X).
   X = 2.

Btw, all that getThirdPeg/3 does is to compute $C := 6 - (A + B)$. 

Ulle Endriss
The Recursive Solution in Prolog

`recSolve(N,A,B)` implements our recursive algorithm to move a tower with `N` disks from location `A` to location `B`:

```prolog
recSolve(N, A, B) :- % to move a tower from A to B
    N =:= 1, % if size N is equal to 1:
    move(A, B). % move single disk from A to B

recSolve(N, A, B) :- % to move a tower from A to B
    N > 1, % if size N is greater than 1:
    getPredecessor(N, N1), % get predecessor N1 of N
    getThirdPeg(A, B, C), % get name of third peg C
    recSolve(N1, A, C), % move tower of size N-1 to C
    move(A, B), % move largest disk to B
    recSolve(N1, C, B). % move tower of size N-1 to B
```

Ulle Endriss
Complexity

How complex is the *Towers of Hanoi* problem? That is, *how long* does it take to solve a game with *n disks*?

**Answer:** depends how fast you can move one disk

**So better ask:** *how many steps* to solve a game with *n disks*?

- $n = 1$: ?
- $n = 2$: ?
- $n = 3$: ?
- ...
Formal Complexity Analysis

Observations: You must move the largest disk (size \( n \)) eventually. You can only move it, if the remaining tower sits somewhere else. So our recursive algorithm is optimal:

(1) move tower of size \( n-1 \) out of the way
(2) move largest disk in place
(3) move tower of size \( n-1 \) on top of it

Let \( f(k) \) be the number of steps needed for a game with \( k \) disks. We know \( f(1) = 1 \) (solving a game with 1 disk takes 1 step).

From our algorithm, for a game with \( n \) disks we immediately get:

\[
 f(n) = f(n-1) + 1 + f(n-1) = 2 \cdot f(n-1) + 1
\]

Not yet helpful for computing, say, \( f(10) \). We want a closed form.
Closed Form

What is $f(n)$, the number of steps to solve a game with $n$ disks?

We have established:

- $f(1) = 1$
- $f(n) = 2 \cdot f(n-1) + 1$ (*)

Examples: $f(2) = 3$, $f(3) = 7$, $f(4) = 15$, $f(5) = 31$, $f(6) = 63$, ...

What we really want:

Claim: $f(n) = 2^n - 1$ for every $n \in \mathbb{N}$.

Proof: Take any $n > 1$. Suppose for a moment that the claim were true for $n-1$, meaning that $f(n-1) = 2^{n-1} - 1$. (**)

But then:

$$f(n) \overset{\text{*}}{=} 2 \cdot f(n-1) + 1 \overset{**}{=} 2 \cdot [2^{n-1} - 1] + 1 = 2^n - 1$$

The claim is true for $n = 1$, as we have seen.

Thus, by induction, it must be true for every $n \in \mathbb{N}$. ✓
Summary

What you’ve learned about today:

• several methods for solving *Towers of Hanoi*
• programming in *Prolog* (well, a bit)

What this really has been about:

• need to find the right *representation* of a problem
• need to *abstract* away from low-level details
• importance of striving for *general* solutions
• importance of formal/mathematical *analysis* of solutions

What next?

• grab a copy of the *handout* on your way out (to have *lunch*)
• be at one of the computer labs at 14:00 for the *practicum*
• submit your *assignment* by 17:00 today